Cache Hierarchy Inspired Compression: a Novel Architecture for Data Streams

Geoffrey Holmes, Bernhard Pfahringer and Richard Kirkby
Computer Science Department
University of Waikato
Private Bag 3105, Hamilton, New Zealand
{geoff, bernhard, rkirkby}@cs.waikato.ac.nz.

Abstract *- We present an architecture for data streams based on structures typically found in web cache hierarchies. The main idea is to build a meta level analyser from a number of levels constructed over time from a data stream. We present the general architecture for such a system and an application to classification. This architecture is an instance of the general wrapper idea allowing us to reuse standard batch learning algorithms in an inherently incremental learning environment. By artificially generating data sources we demonstrate that a hierarchy containing a mixture of models is able to adapt over time to the source of the data. In these experiments the hierarchies use an elementary performance based replacement policy and unweighted voting for making classification decisions.*

Keywords: Data streams, classification, cache hierarchy.

1 Introduction

Conventional data mining algorithms operate in an environment where a model (for example, a set of rules) is induced from a training set of data instances. This training set is available in its entirety from the outset. The learning algorithms that produce a model from this data typically load all of the data into main memory and then access each instance one at a time. This methodology is often referred to as batch learning. A consequence of this approach is that very few incremental algorithms have been developed.

The data stream model treats the training data as a (possibly infinite) *stream.* In this context there is no possibility of fitting all the data into memory. The model inevitably places constraints on what can be achieved and scaling an algorithm to conform to a data stream is a significant challenge. The requirements are: the need to process instances one at a time with only a glance at the instance (instances cannot be stored); only use available memory (we cannot swap to disk as this would take too long); process an instance in a limited amount of time (we cannot afford to fall behind the rate at which instances arrive); and finally, be ready to make predictions at *any* time.

In such a context it is extremely difficult to fully process a single instance at a time. One approach to staying in touch with the stream is to accumulate statistics and then make decisions only when those statistics reach critical points, as in Hoeffding trees [1]. An alternative approach is to slowly build up a knowledge of the stream by constructing models from accumulations of instances [2]. Both strategies can be accommodated in the architecture described here.

In the field of Data Communications, web proxy caches are used to enhance the scalability and performance of the web by reducing bandwidth demand and increasing the response time for popular documents. The proxies are often organised into hierarchies as in the squid system [3]. In squid the hierarchy is organized in a parent/child and sibling arrangement. When a cache requests an object from its parent, and the parent does not have the object in its cache, the parent fetches the object, caches it, and delivers it to the child. In addition to the parent-child relationships there are siblings which are caches at the same level in the hierarchy, provided to distribute cache server load.

In a streaming context it would be advantageous to distribute algorithmic load (classification, regression, clustering or association rule learning) through some form of sibling structure. The notion of parent/child caches is harder to reconcile as the objectives of web caching and data mining are somewhat different. A parent could be used to contain older and wiser models, those that have successfully survived some form of culling process over time.

Alternatively, parents could be viewed as the agents in a system that pass on the best models they find for use by their children.

In data streaming and web caching, training and testing have to occur simultaneously. The web cache hierarchy fills up by fetching objects not present in the cache until it is full. Once full, an object replacement policy is needed to keep the hierarchy up to date with the needs of its user community. Replacement policies are typically designed to optimise a performance metric such as hit rate (the number of successful retrievals from the cache) or byte hit rate if object size is a more important factor. Popular

policies include Least Recently Used (LRU), replace the newest object with the oldest object, Least Frequently Used (LFU) replace the newest object with the least frequently requested and so on.

In order to replicate such a system in a data stream context, we need to design a hierarchy, fill it with models and implement a replacement policy so that the set of models contained in the hierarchy is current and tuned to the performance metric that most suits the task at hand. It is not likely, therefore, that LRU style policies would be directly adopted.

In this paper we explore the construction of a web cache inspired architecture for mining data streams. The intention is to demonstrate the efficacy of the technique rather than explore the many options that could be tried in terms of numbers of levels, replacement policies, mining algorithms and so on.

The paper is structured as follows. In Section 2 we present the general architecture for the cache hierarchy inspired compression system (CHIC). We describe the range of policy decisions that need to be made and the choices made for this paper. Section 3 discusses the generation of data and design of experiments to test the method. Section 4 contains a description of an instance of the architecture using classification and experimental results on a collection of real and artificial datasets. Section 5 discusses related work on committee and non-committee based classifiers for data streams. Section 6 summarizes the contributions made in this paper.

2 CHIC

Cache Hierarchy Inspired Compression (CHIC) is a system for mining data streams based on the idea of caching models and replacing them continually as the stream is processed. The hierarchy is constructed from a number of levels that have four different functions.

It is useful to imagine that each level of the hierarchy is a buffer of a fixed size. Level zero is unique in that it contains instances whereas all other levels contain models.

Level one in the hierarchy is where all the work is done. Models are continually constructed from the level zero buffer which is refreshed once a model is constructed and housed in level one. This process is repeated until level one is full.

Once level one is full a model is selected for promotion to level two. Once selected, the level one buffer is cleared of models and the process resumes. Once level two is full a model is similarly selected for promotion to level three and the process resumes. This replacing of models continues all the way up the hierarchy until the topmost level is reached. At this level a model must be selected for ejection from the hierarchy so that the process can continue to build models from the stream. Of course, it would be possible to freeze the hierarchy once full so as to build a meta level analyser slowly over many instances.

In this paper we choose to reject models from the topmost level in order to keep the hierarchy up to date, as a web cache would. In theory this would enable concept drift to be tracked. Concept drift is another major topic in mining data streams but we do not attempt to solve this problem in this paper.

The overall aim of the system is to build a meta level algorithm from the models in levels one upward. The general methodology for constructing such a model containing N levels is as follows:

- *Buffering*: At level zero fill a buffer with data.
- *Model Building*: At level one build models and place them in the buffer, when full promote models and flush the buffer.
- *Containers*: At levels two to N-1 adopt models from the level below, when full promote models and flush the buffer.
- *Topmost*: Adopt models from level N-1 but also discard models so that new ones can be entered.

This basic algorithm requires policies for the promotion of models between levels and for discarding models from the topmost level. It also leaves open the question of what types of model to generate. The hierarchy could be homogeneous or, as in this paper, it could contain a mixture of models.

Although the algorithm appears to simply add models to levels it would be possible and desirable to consider merging models in some applications, for example, clustering.

It is important to note that the CHIC architecture is an instance of the general wrapper idea [11], which allows us to use any arbitrary batch learning algorithm for learning from data streams, which is an inherently incremental problem. Other standard stream learning algorithms are usually purpose-built incremental versions of standard batch learning algorithms (e.g. Hoeffding trees [1]). Using the CHIC architecture we can employ any off-the-shelf algorithm avoiding expensive development work.

In order to arrive at a decision with the meta level model it is necessary to decide how to vote the models. Given the hierarchical structure it would make sense to weight the votes on the position in the hierarchy rather than treat everything as equal. If efficiency is an issue then decisions could be made on the basis of votes at only the topmost level using the argument that these are the best surviving models. In this paper we restrict attention to unweighted voting, where every model has an equal vote.

The options for promoting models are manifold. Here we only analyse performance based selection.

3 Experimental Design

In this paper we aim to show that the hierarchy of models is a useful architecture for mining data streams. To this end we do not attempt to find optimal settings, policies or combinations of methods that demonstrate superiority over others.

We have created the simplest manifestation of the architecture. First, we use classification for testing as it is the easiest task to evaluate. Second, we learn a mixture of models at levels one and two. One advantage of this hierarchy is that the selection mechanism used to populate the container and topmost levels will then adapt the mixture of models at the base to the likely data source.

If we run with a single algorithm such as a tree learner then performance on data generated from a Bayesian source will be poor, and vice versa. If the lower levels contain a mixture of tree and Bayesian learners then those most suited to the data source will be chosen.

The size of the data buffer, often referred to as chunk size, at level zero can have an impact on performance. If it is too small then the classifiers may not have the capacity to generalize, too large and the cost of the level one buffer will be too high in terms of keeping up with the stream. In this paper we use a fixed size data buffer of 1000 instances which we found to be a good compromise in previous work on large datasets [2].

3.1 Evaluation

Two kinds of evaluation are required for this setting. First, models must be evaluated against their peers to see if they need promoting or deleting within/from the hierarchy. Second, the hierarchy itself must be evaluated as it evolves.

Peer analysis of models is achieved by analyzing performance on the next buffer of input. Model performance statistics are incrementally updated so that performance can be assessed over longer periods of time than a single buffer. If this is not done then the estimates of model performance will be too unstable to be of any use as models need as much time as possible to show what they can do.

To assess the performance of the hierarchy as a whole we employ a general evaluation framework that uses a *first test then train* approach to an instance. This is to ensure that the algorithmic basis for *any* mining algorithm has been engineered to suit a real data stream. By testing an instance before using it for training we are primarily insisting on incrementality. Performance is also implied in that a good model will return a prediction in real time only if the model is not too elaborate and only if it fits in memory. Thus, a model that fails to comply with the paradigm will eventually do so by taking too long to form a prediction. The evaluation methodology is as follows:

- Initialize working model M to an empty model
- While more instances are available in the stream
- Retrieve next instance I from the stream
- Classify I using M
- Update accuracy statistics S, storing if desired
- Increment M by training with I

A snapshot can be taken at any point during the evaluation process. These snapshots can be plotted graphically to produce learning curves. Typically the statistic of most interest is related to the percentage of instances that were correctly classified. The resulting curve is smooth between snapshots because as the number of processed instances increases, the influence of a single classification/misclassification on the overall percentage becomes smaller.

As data order can be a source of concern in data stream classification, the evaluation method presented here can be extended to average over many different orderings. The evaluation algorithm is repeated multiple times, each time the data is presented in a different order (or in the case of data generators, an independent set of examples). The snapshots at each data point are collected together and averaged between runs. This extension allows analysis of the variance of the performance over several data orders. The results presented later in Section 4 (Figure 3) represent averages of five separate orderings. It should be noted that in all cases there was very little difference in performance between runs. Thus for these datasets data order does not appear to be a major issue.

In terms of an actual testing mechanism for the hierarchy, unweighted voting was used for making classification decisions. Again this was chosen for simplicity.

3.2 Data Sources

The data sets used in this paper consist of three synthetically generated and one real dataset.

3.2.1 Random Naïve Bayes

Random Naive Bayes data is generated by first randomly assigning a weight to each class, and then randomly assigning properties to the attributes. In the case of nominal attributes, each nominal label is given a random weight per class. In the case of numeric attributes a Gaussian distribution (randomly chosen mean and standard deviation) is assigned per class.

To generate new instances, a class is first randomly chosen with likelihood in proportion to the pre-assigned class weights. Then for each attribute a value is drawn at random according to the pre-assigned distribution of values for the particular class.

The Naive Bayes data reported here has five classes, five nominal attributes (with five labels each) and five numeric attributes.

3.2.2 Random tree

The random tree data generator works by first constructing a random tree—at each node an attribute is chosen at random to split the data, and if the attribute is numeric, a split point is randomly chosen too. This process is applied recursively until the desired depth is reached, at which point a class label is randomly assigned. If a nominal attribute is used to split then it cannot be reused in the subtree below it. There is an optional parameter for a depth at which below there is a random chance of a node becoming a leaf before the maximum depth is hit. Once the random tree is complete, instances are generated by randomly assigning values to attributes, then traversing the tree to assign the class.

The random tree dataset is generated by a tree that is five nodes deep, with leaves starting at level three. It has five nominal attributes (with five labels each), five numeric attributes and five classes.

3.2.3 Random RBF

Random RBF data is generated by first creating a random set of centers for each class. Each center is randomly assigned a weight, a central point per attribute, and a standard deviation. To generate new instances, a center is chosen at random taking the weights of each center into consideration. Attribute values are randomly generated and offset from the center, where the overall vector has been scaled so that its length equals a value sampled randomly from the Gaussian distribution of the center. The particular center chosen determines the class of the instance.

For this paper the data contains only numeric attributes as it is non-trivial to include nominal values. The random RBF data is generated from 50 centers, has ten numeric attributes and five classes.

3.2.4 Forest Covertype

One of the largest datasets containing real-world data in the UCI repository [4] is the Forest Covertype dataset. This consists of 581,012 instances, 10 numeric attributes, 44 binary attributes and 7 classes.

4 Results

4.1 Specific CHIC Architecture

We conducted experiments for a hierarchy consisting of six layers, with 16 models per layer. The data is divided into chunks of size 1000 for training the models. On the first level, four instantiations of four different learning algorithms are used to train models. The four learning algorithms are naive Bayes, decision trees, linear support vector machines and the support vector machine with RBF kernel [5]. All algorithms use default parameter settings, with the exception that for every new RBF SVM the gamma parameter is randomly selected from the set {1, 4, 16, 64}. Four methods were used to provide differing bias.

The mechanics of building models follows the outline of the general architecture with some special features designed to capture the data source. The evolution of the models on random tree data for the first 13,000 instances is depicted in Figure 1. Here N denotes naïve Bayes, J decision trees, L linear support vector machines and R nonlinear support vector machines, and – denotes an empty space.

To begin, 1000 instances are read into the level zero buffer, and one model of each of the four methods is installed at level one. The next 3000 instances then leads to the complete population of level one with four models for each method. The next 1000 instances then generates four more models. At this stage two actions are undertaken. First, the worst four models (one from each group) are discarded to open up space at level one. Second, the best four models are promoted to level two. This gives the picture after 5000 instances. Four models from the next 1000 instances then fill the spaces created by the previous deletions. This process continues until level two is fully populated (after 12,000 instances).

Once level two is full, however, the population of higher levels is achieved by promoting only the best models. Thus, after 13,000 instances the four true worst models from level two are deleted (i.e. not one from each group) and the best four models, again from any group, are promoted. This frees eight spaces at level two.

At level one the process continues as before, four models are deleted (one from each group) and four are promoted to level two, again one from each group. The general idea is to maintain the spread of methods at the base levels and allow the higher levels to best reflect the source of the data. In Figure 1 this is apparent immediately since the decision trees are the first selected.

Levels three, four and five operate on the same basis of deleting and promoting only the best models. The topmost level has no space to promote models to and so only deletes the worst four. Figure 2 shows the final configurations of the six levels for each of the data sources.

Figure 2 shows how CHIC is able to adapt to the data source by promoting the best performing models. For random tree data the higher levels are populated with decision tree models, naïve Bayes models for the naïve Bayes source and nonlinear support vector machines for the RBF data. The most interesting configuration is the one pertaining to the real dataset *covertype*. Here a mixture of linear support vector machines and decision trees.

The performance of the evolving configurations for each of the data sources is depicted in Figure 3. Each of the synthetic datasets processed one million instances. Each graph in Figure 3 shows a common pattern of learning quickly and then flattening out after approximately 100,000 instances. For the random tree and naïve Bayes data sets the concepts are learned almost perfectly. Clearly, the RBF data is a greater challenge.

after 1000 instances: N---J---L---R---	after 2000 instances: NN--JJ--LL--RR--
after 3000 instances: NNN-JJJ-LLL-RRR-	after 4000 instances: NNNNJJJJLLLLRRRR
after 5000 instances: N-NNJJ-JL-LLR-RR NJLR----------	after 6000 instances: NNNNJJJJLLLLRRRR NJLR----------
after 7000 instances: NNN-J-JJL-LLR-RR NJLRNJLR--------	after 8000 instances: NNNNJJJJLLLLRRRR NJLRNJLR--------
after 9000 instances: NN-N-JJJLL-L-RRR NJLRNJLRNJLR----	after 10000 instances: NNNNJJJJLLLLRRRR NJLRNJLRNJLR----
after 11000 instances: -NNN-JJJLLL-RR-R NJLRNJLRNJLRNJLR	after 12000 instances: NNNNJJJJLLLLRRRR NJLRNJLRNJLRNJLR
after 13000 instances: NN-NJJ-JLL-LRR-R NNLJNLLRN-L-N-L- JJJJ----------	

Figure 1. Evolution of models on random tree data

It is important to note that all models are able to update their statistics for all instances processed. This keeps each model fresh in the hierarchy but does impose a performance penalty which can slow the down the architecture's ability to process the data stream.

random tree	**random naive Bayes**
N-NN-JJJLL-LR-RR	-NNNJJ-JLLL-R-RR
NNNNLNNLJLRNNJLR	NJJJLLLLRNJJJLLR
JJJJJNNLJJ---JL-	NNNNNJJNJJ-NN---
JJJJJJJJJJJJJJJ	NNNNNNNNNNNNNNNN
JJJJ--JJ-JJJJJ-J	NNNNNNNN---NNNN-
JJJJJJJJJJJJJJ	NNNNNNNNNNNNNNNN

random RBF	**covertype**
NNN--JJJLL-LRR-R	N-NN-JJJ-LLLR-RR
NNJJLRNJNNJJNJLR	NNJJJRJLNRJ---R-
RRJRJJRRRRJRRJJJ	LJJLLJLJLJL--N--
RRRRRRRRRRRRRRRR	LLLLLLLLLLLLJLLJ
RRRRRRRRRRRRRRRR	LLLLLLLLJL-J-L--
RRRRRRRRRRRRRRRR	LLLLLLLLLLLLLLLL

Figure 2. Final level configurations

This penalty can be reduced by only updating models every so often rather than on a per instance basis. We have not experimented with this approach. It is likely that this would not perform as well as the approach described here but may be necessary in applications involving high speed data streams.

5 Related Work

In general, work on data streams proceeds in a number of areas. Mining data streams, query processing, modelling, and the detection of concept drift have dominated the literature. Concept drift in particular is a hot topic. The use of multiple model (ensemble) techniques over single models is also gaining acceptance due to their increased performance, efficiency and potential for parallelization [6].

There are clear parallels between the hierarchical model presented here and bagging [7]. In a data stream context there is no need to sample with replacement as there is not a limited supply of data. Bagging is particularly robust to noise but generally not as well performing as boosting.

Boosting in this context could be achieved by boosting individual models or by building a first model at level one and then using it to weight incoming instances for the second model (at level one). This procedure would be repeated until the level was full.

Work of a similar vein for data sources of fixed size has been reported in [8] who propose an incremental version of AdaBoost. Their method retains a fixed-size window of weak classifiers that contains the *k* most recently built classifiers. This makes the method applicable to large datasets in terms of memory and time requirements. However, it remains unclear how an appropriate value for *k* can be determined.

Street and Kim [9] propose a variant of bagging for incremental learning based on data chunks that maintains a fixed-size committee. In each iteration it attempts to identify a committee member that should be replaced by the model built from the most recent chunk of data.

Oza and Russell [10] propose incremental versions of bagging and boosting that require the underlying weak learner to be incremental. The method is of limited use for large datasets if the underlying incremental learning algorithm does not scale linearly in the number of training instances. Unfortunately, the time complexity of most incremental learning algorithms is worse than linear.

6 Conclusion

We have presented a novel architecture for processing data streams. The architecture operates much like a web cache in that models are propagated from one level to another on the basis of a replacement policy. This

architecture is applicable to classification, regression, clustering and association rule learning.

By employing a mixtures of methods so that the most appropriate become more plentiful within the higher levels of the architecture it is possible to demonstrate that it is possible for the architecture to adapt to the source of the data.

There are many possibilities for future work on this project. There are many ways in which the architecture could be configured. Prediction methodology could also be varied by using, for example, only the wisest models (models in the topmost level) to make predictions rather than using the entire configuration which can be slow.

References

[1] P. Domingos and G. Hulten, "Mining high-speed data streams", Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 71-80, 2000.

[2] E. Frank, G. Holmes, R. Kirkby and M. Hall, "Racing Committees for Large Datasets", Proc. International Conference on Discovery Science, 153-164, 2002.

[3] D. Wessels, *Squid: The Definitive Guide*, O'Reilly, 2004.

[4] C.L. Blake and C.J. Merz, "UCI Repository of machine learning databases", http://www.ics.uci.edu/~mlearn/MLRepository.html, University of California, Irvine, Dept. of Information and Computer Sciences, 1998.

[5] Witten, I.H., Frank, E. *Data mining: practical machine learning tools and techniques.* (second ed). Morgan Kaufmann, San Francisco, CA, 2005.

[6] H. Wang, W. Fan, P. Yu and Jiawei Han, "Mining Concept-Drifting Data Streams Using Ensemble Classifiers", Proc. Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD 2003) 2003.

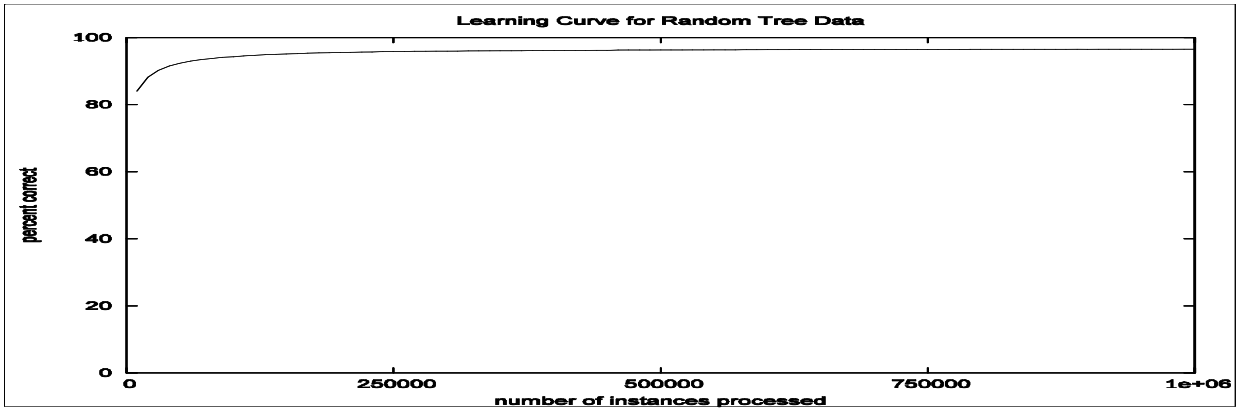
[7] L. Breiman, "Bagging Predictors", Machine Learning Journal,Vol 24, No. 2, 123-140, 1999.

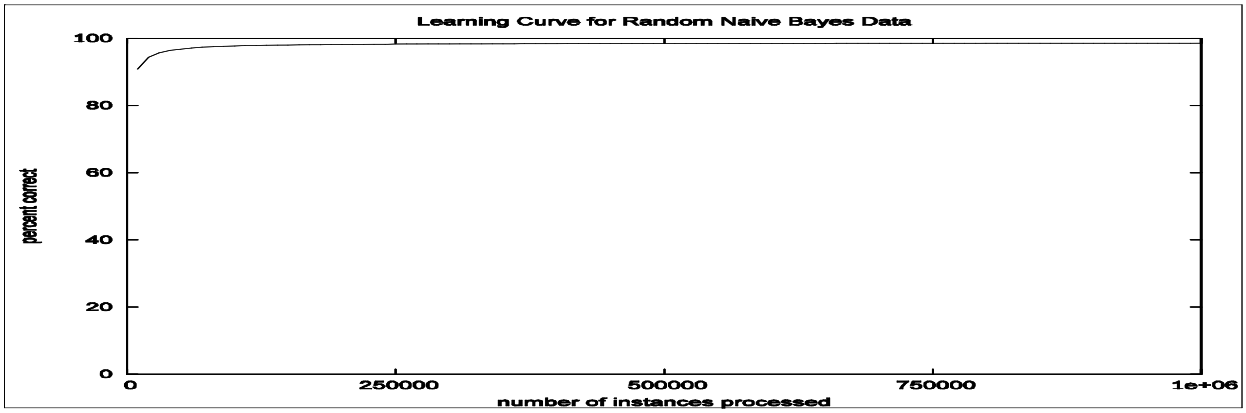
[8] W. Fan, S. J. Stolfo and J. Zhang, "The Application of AdaBoost for Distributed, Scalable and On-Line Learning", Proc. 5th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, 362-366, 1999.

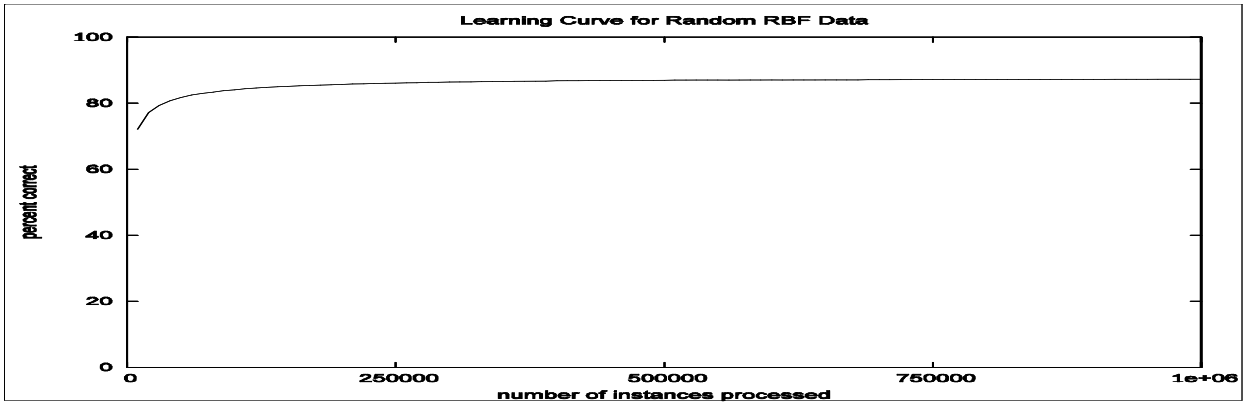
[9] W. Street and Y. Kim, "A streaming ensemble algorithm (SEA) for large-scale classification", Proc. 7th ACM SIGKDD Int. Conf. on Knowledge Discovery in Databases and Data Mining, 377-382, 2001.

[10] N. Oza and S. Russell, "Experimental Comparisons of Online and Batch Versions of Bagging and Boosting", Proc. 7th ACM SIGKDD Int. Conf. on Knowledge Discovery in Databases and Data Mining, 359-364, 2001.

[11] R. Kohavi and G.H. John, "The wrapper approach" in "Feature Extraction, Construction and Selection: A Data Mining Perspective", edited by H. Liu and H. Motoda, Kluwer Academic, 1998.







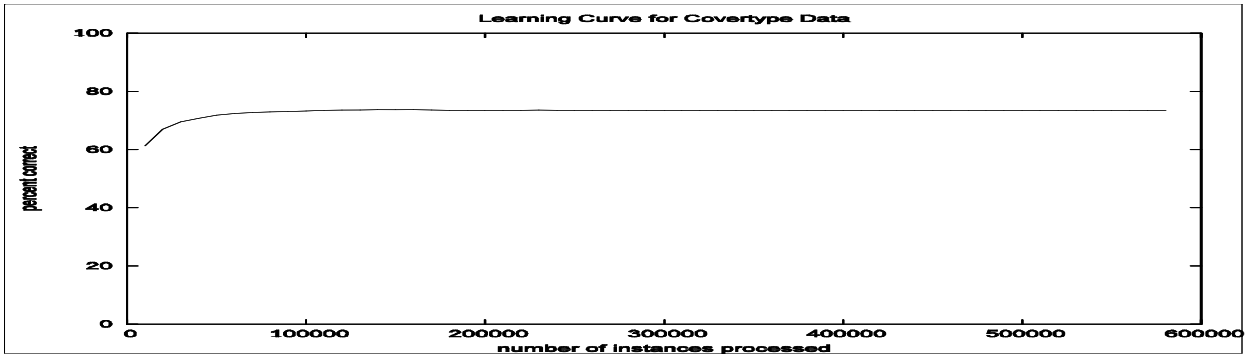


Figure 3. Learning curves for data sets